Apr 28, 00 12:47 **Recursive.m3** Page 1/1

```
MODULE Recursive EXPORTS Main ;

IMPORT IO ;


PROCEDURE BinKoef(n, m : INTEGER) : INTEGER =
BEGIN
    IF m = 0 THEN
        RETURN 1 ;
    ELSIF n = m THEN
        RETURN 1 ;
    ELSE
        RETURN BinKoef(n-1, m-1) + BinKoef(n-1, m) ;
    END ;
END BinKoef ;


CONST N = 10 ;

BEGIN
    FOR n := 0 TO N DO
        FOR m := 0 TO n DO
            IO.PutInt(BinKoef(n, m)) ;
            IO.Put(" ") ;
        END ;
        IO.Put("\n") ;
    END ;
END Recursive .
```

May 03, 00 18:15 **NonRecursive.m3** Page

```
MODULE NonRecursive EXPORTS Main ;

(* Variante 1: quadratische Speicherplatzkomplexitaet *)

IMPORT IO ;


CONST N = 10 ;

VAR a : ARRAY [0..N] OF ARRAY [0..N] OF INTEGER ;

BEGIN
    FOR n := 0 TO 10 DO
        FOR m := 0 TO n DO
            IF m = 0 THEN
                a[n][m] := 1 ;
            ELSIF n = m THEN
                a[n][m] := 1 ;
            ELSE
                a[n][m] := a[n-1][m-1] + a[n-1][m] ;
            END ;
            IO.PutInt(a[n][m]) ;
            IO.Put(" ") ;
        END ;
        IO.Put("\n") ;
    END ;
END NonRecursive .
```

May 03, 00 18:15 **NonRecursive2.m3** Page 1/1

```
MODULE NonRecursive2 EXPORTS Main ;

(* Variante 2: weniger gut lesbar, dafuer bessere
 *  Speicherplatzausnutzung O(n) *)

IMPORT IO ;


CONST N = 10 ;

VAR a : ARRAY [0..N] OF INTEGER ;

BEGIN
    FOR n := 0 TO 10 DO
        FOR m := n TO 0 BY -1 DO
            IF m = 0 THEN
                a[m] := 1 ;
            ELSIF n = m THEN
                a[m] := 1 ;
            ELSE
                a[m] := a[m-1] + a[m] ;
            END ;
            IO.PutInt(a[m]) ;
            IO.Put(" ") ;
        END ;
        IO.Put("\n") ;
    END ;
END NonRecursive2 .
```

May 03, 00 18:08 **output.txt** Page

```
1
1  1
1  2  1
1  3  3  1
1  4  6  4  1
1  5  10  10  5  1
1  6  15  20  15  6  1
1  7  21  35  35  21  7  1
1  8  28  56  70  56  28  8  1
1  9  36  84  126  126  84  36  9  1
1  10  45  120  210  252  210  120  45  10  1
```

Apr 28, 00 12:47 **Recursive.m3** Page 1/1

```
MODULE Recursive EXPORTS Main ;

IMPORT IO ;


PROCEDURE BinKoef(n, m : INTEGER) : INTEGER =
BEGIN
    IF m = 0 THEN
        RETURN 1 ;
    ELSIF n = m THEN
        RETURN 1 ;
    ELSE
        RETURN BinKoef(n-1, m-1) + BinKoef(n-1, m) ;
    END ;
END BinKoef ;


CONST N = 10 ;

BEGIN
    FOR n := 0 TO N DO
        FOR m := 0 TO n DO
            IO.PutInt(BinKoef(n, m)) ;
            IO.Put(" ") ;
        END ;
        IO.Put("\n") ;
    END ;
END Recursive .
```

May 03, 00 18:15 **NonRecursive.m3** Page

```
MODULE NonRecursive EXPORTS Main ;

(* Variante 1: quadratische Speicherplatzkomplexitaet *)

IMPORT IO ;


CONST N = 10 ;

VAR a : ARRAY [0..N] OF ARRAY [0..N] OF INTEGER ;

BEGIN
    FOR n := 0 TO 10 DO
        FOR m := 0 TO n DO
            IF m = 0 THEN
                a[n][m] := 1 ;
            ELSIF n = m THEN
                a[n][m] := 1 ;
            ELSE
                a[n][m] := a[n-1][m-1] + a[n-1][m] ;
            END ;
            IO.PutInt(a[n][m]) ;
            IO.Put(" ") ;
        END ;
        IO.Put("\n") ;
    END ;
END NonRecursive .
```

May 03, 00 18:15 **NonRecursive2.m3** Page 1/1

```
MODULE NonRecursive2 EXPORTS Main ;

(* Variante 2: weniger gut lesbar, dafuer bessere
 *  Speicherplatzausnutzung O(n) *)

IMPORT IO ;


CONST N = 10 ;

VAR a : ARRAY [0..N] OF INTEGER ;

BEGIN
    FOR n := 0 TO 10 DO
        FOR m := n TO 0 BY -1 DO
            IF m = 0 THEN
                a[m] := 1 ;
            ELSIF n = m THEN
                a[m] := 1 ;
            ELSE
                a[m] := a[m-1] + a[m] ;
            END ;
            IO.PutInt(a[m]) ;
            IO.Put(" ") ;
        END ;
        IO.Put("\n") ;
    END ;
END NonRecursive2 .
```

May 03, 00 18:08 **output.txt** Page

```
1
1  1
1  2  1
1  3  3  1
1  4  6  4  1
1  5  10  10  5  1
1  6  15  20  15  6  1
1  7  21  35  35  21  7  1
1  8  28  56  70  56  28  8  1
1  9  36  84  126  126  84  36  9  1
1  10  45  120  210  252  210  120  45  10  1
```

May 18, 00 18:11 **Main.m3** Page 1/1

```
MODULE Main EXPORTS Main ;

IMPORT IO ;


PROCEDURE BinarySearch(READONLY a: ARRAY OF INTEGER ; v : INTEGER) : BOOLEAN =

    PROCEDURE BinSearch(i, j : INTEGER) : BOOLEAN =
    VAR m : INTEGER ;
    BEGIN
        IF i <= j THEN
            m := (i + j) DIV 2 ;
            IF v = a[m] THEN
                RETURN TRUE ;
            ELSIF v < a[m] THEN
                RETURN BinSearch(i, m-1) ;
            ELSE (* v > a[m] *)
                RETURN BinSearch(m+1, j) ;
            END ;
        ELSE
            RETURN FALSE ;
        END ;
    END BinSearch ;

BEGIN
    RETURN BinSearch(FIRST(a), LAST(a)) ;
END BinarySearch ;


CONST data = ARRAY OF INTEGER {
  -798, -611, -495, -299, -222, -125, -105, -100, -89, -34, -30, -8,
  -5, -3, 2, 10, 13, 18, 20, 100, 105, 121, 200, 300, 450, 500, 501} ;

BEGIN
    FOR v := -1000 TO 1000 DO
        IF BinarySearch(data, v) THEN
            IO.PutInt(v) ;
            IO.Put("\n") ;
        END ;
    END ;
END Main .
```

May 22, 00 18:39 **output.txt** Page

```
-798
-611
-495
-299
-222
-125
-105
-100
-89
-34
-30
-8
-5
-3
2
10
13
18
20
100
105
121
200
300
450
500
501
```

May 30, 00 9:48 **Main.m3** Page 1/1

```
MODULE Main ;

IMPORT IO ;
IMPORT LinkedList AS List ;


(* Liste sortieren nach der Methode "Sortieren durch Einfuegen". *)

PROCEDURE SortList() =
VAR v : INTEGER ;
BEGIN
    List.ToFirst() ;
    LOOP
        (* Vorwaerts bis zum naechsten fehlstehenden Element. *)
        REPEAT
            v := List.Get() ;
            List.Forth() ;
            IF List.IsAfter() THEN
                RETURN ; (* Ende der Liste erreicht -> fertig *)
            END ;
        UNTIL List.Get() < v ;

        (* Element entfernen, ... *)
        v := List.Get() ;
        List.Delete() ;
        (* ... zurueck bis zur Einfuegeposition, ... *)
        REPEAT
            List.Back() ;
        UNTIL  List.IsBefore() OR List.Get() < v ;
        List.Forth() ;
        (* ... und einfuegen. *)
        List.InsertBefore(v) ;
    END ;
END SortList ;


PROCEDURE WriteList() =
BEGIN
    List.ToFirst() ;
    WHILE NOT List.IsAfter() DO
        IO.PutInt(List.Get()) ;
        IO.Put("\n") ;
        List.Forth() ;
    END ;
END WriteList ;

CONST data = ARRAY OF INTEGER {
  18, -5, -8, -3, 121, -30, 100, -89, 20, -100, 10, 105, -34, 2, -105,
  200, 300, -611, 500, -222, -299, -495, -125, 501, -798, -30, 450, 13} ;


BEGIN
    List.Init() ;
    FOR i := FIRST(data) TO LAST(data) DO
        List.Append(data[i]) ;
    END ;

    SortList() ;
    WriteList() ;
END Main .
```

May 25, 00 11:55 **output.txt** Page

```
-798
-611
-495
-299
-222
-125
-105
-100
-89
-34
-30
-30
-8
-5
-3
2
10
13
18
20
100
105
121
200
300
450
500
501
```

```
MODULE Main ;

IMPORT IO, Tick ;
FROM Aufgabe35 IMPORT Random ;


(* Verbessertes QuickSort: Laesst Teilfolgen mit bis zu k Elementen
   unsortiert.  Verbliebene Unordnung wird mit InsertionSort bereinigt. *)


PROCEDURE QuickSort(VAR a : ARRAY OF INTEGER ; k : CARDINAL) =

    PROCEDURE Partition(l, r : INTEGER) : INTEGER =
    VAR v, t, i, j : INTEGER ;
    BEGIN
        i := l - 1 ;
        j := r ;
        v := a[r] ; (* waehle Pivot-Element *)
        REPEAT
            REPEAT INC(i) UNTIL a[i] >= v ;
            REPEAT DEC(j) UNTIL a[j] <= v ;
            t := a[i] ; a[i] := a[j] ; a[j] := t ;
        UNTIL j <= i ; (* Zeiger kreuzen *)
        a[j] := a[i] ; a[i] := a[r] ; a[r] := t ;
        RETURN i ;
    END Partition ;

    PROCEDURE QS(l, r : INTEGER) =
    VAR i : INTEGER ;
    BEGIN
        IF r > l + k THEN (* <- einziger Unterschied zum normalen QuickSort *)
            i := Partition(l, r) ;
            QS(l, i-1) ;
            QS(i+1, r) ;
        END ;
    END QS ;

BEGIN
    QS(1, LAST(a)) ;
END QuickSort ;


PROCEDURE InsertionSort(VAR a : ARRAY OF INTEGER) =
VAR j, t : INTEGER ;
BEGIN
    FOR i := 1 TO LAST(a) DO
        t := a[i] ;
        j := i ;
        WHILE t < a[j-1] DO
            a[j] := a[j-1] ;
            DEC(j) ;
        END ;
        a[j] := t ;
    END ;
END InsertionSort ;


PROCEDURE CombinedSearch(VAR a : ARRAY OF INTEGER ; k : CARDINAL) =
BEGIN
    a[0] := 0 ; (* Sentinel *)
    QuickSort(a, k) ;
    InsertionSort(a) ;
END CombinedSearch ;
```



```
(* Hauptprogramm fuehrt einige Laufzeitmessungen zur Bestimmung der
   optimalen Konstanten k durch *)

CONST n = 1000000 ;
      k_max = 200 ;
      num_tests = 10 ;

VAR data, work : ARRAY [0..n] OF INTEGER ;
    t1, t2 : Tick.T ;
    times := ARRAY [0..k_max] OF REAL {0.0,..} ;

BEGIN
    FOR t := 1 TO num_tests DO
        (* Feld mit Zufallszahlen erzeugen *)
        FOR i := 1 TO n DO
            data[i] := Random(1000000) ;
        END ;

        (* Laufzeitmessung fuer unterschiedliche Werte von k *)
        FOR k := 0 TO 150 BY 5 DO
            (* erzeuge Arbeitskopie des Feldes, damit jeder Aufruf die
               gleichen Eingabedaten hat *)
            work := data ;

            t1 := Tick.Now() ;
            CombinedSearch(work, k) ;
            t2 := Tick.Now() ;

            IO.PutInt(k) ; IO.Put("\t") ;
            IO.PutReal(FLOAT(Tick.ToSeconds(t2 - t1))) ; IO.Put("\n") ;
            times[k] := times[k] + FLOAT(Tick.ToSeconds(t2 - t1)) ;
        END ;
        IO.Put("\n") ;
    END ;

    (* Mittelwerte ausgeben *)
    IO.Put("# averages:\n") ;
    FOR k := 0 TO 150 BY 5 DO
        IO.PutInt(k) ;
        IO.Put("\t") ;
        IO.PutReal(times[k] / FLOAT(num_tests)) ;
        IO.Put("\n") ;
    END ;
END Main .
```

```
MODULE Main ;

IMPORT IO ;
FROM Aufgabe35 IMPORT Random ;

(* Bestimmung des Medians einer Zahlenfolge nach einer
   Divide-and-Conquer-Strategie.  Idee:
   1. Partitioniere das Feld genau wie bei QuickSort.
   2. Je nachdem ob das Pivot-Element links oder rechts von der Mitte des
      Feldes zu liegen kommt, durchsuche rekursive entweder das linke
      *oder* das rechte Teilfeld. *)

PROCEDURE Median(VAR a : ARRAY OF INTEGER) : INTEGER =

    VAR m : INTEGER ; (* Position des Medians im sortierten Feld *)

    PROCEDURE Partition(l, r : INTEGER) : INTEGER =
    VAR v, t, i, j : INTEGER ;
    BEGIN
        i := l - 1 ;
        j := r ;
        v := a[r] ; (* waehle Pivot-Element *)
        REPEAT
            REPEAT INC(i) UNTIL a[i] >= v ;
            REPEAT DEC(j) UNTIL a[j] <= v ;
            t := a[i] ; a[i] := a[j] ; a[j] := t ;
        UNTIL j <= i ; (* Zeiger kreuzen *)
        a[j] := a[i] ; a[i] := a[r] ; a[r] := t ;
        RETURN i ;
    END Partition ;

    PROCEDURE Med(l, r : INTEGER) : INTEGER =
    VAR i : INTEGER ;
    BEGIN
        IF r > l THEN
            i := Partition(l, r) ;
            IF i = m THEN
                RETURN a[i] ;
            ELSIF i > m THEN
                RETURN Med(l, i-1) ;
            ELSE
                RETURN Med(i+1, r) ;
            END ;
        ELSE
            RETURN a[m] ;
        END ;
    END Med ;

BEGIN
    a[0] := 0 ; (* Sentinel *)
    m := LAST(a) DIV 2 ;
    RETURN Med(1, LAST(a)) ;
END Median ;


CONST n = 50 ;
VAR data : ARRAY [0..n] OF INTEGER ;
BEGIN
    FOR i := 1 TO n DO
        data[i] := Random(1000) ;
        IO.PutInt(data[i]) ; IO.Put("\n") ;
    END ;

    IO.Put("Median:") ;
    IO.PutInt(Median(data)) ; IO.Put("\n") ;
END Main .
```



```
963
526
992
774
939
53
589
338
71
150
270
402
263
665
452
881
558
695
779
771
581
951
132
866
807
821
355
180
263
567
142
785
153
883
797
704
345
6
270
166
493
972
366
750
92
775
508
704
364
883
Median : 558
```

```
INTERFACE Aufgabe35 ;

(* Generiert Pseudozufallszahl zwischen 1 und n inklusive *)
PROCEDURE Random(n : CARDINAL) : CARDINAL ;

END Aufgabe35 .
```



```
MODULE Aufgabe35 ;

CONST a = 16807 ;
      m = 2147483647 ;
      q = m DIV a ;
      r = m MOD a ;

VAR   z := 42 ;


PROCEDURE Random(n : CARDINAL) : CARDINAL =
BEGIN
    z := a * (z MOD q) - r * (z DIV q) ;
    IF z < 0 THEN
        z := z + m ;
    END ;
    RETURN z MOD n + 1 ;
END Random ;


BEGIN
END Aufgabe35 .
```

```
MODULE Main ;

IMPORT IO ;
FROM Aufgabe35 IMPORT Random ;

(* Variante von BucketSort, die in situ arbeitet, dafuer aber nicht
   stabil ist.  Ausser dem Histogramm wird nur Speicher O(1)
   benoetigt.  Idee:
   1. Erzeuge Histogramm
   2. Summiere Histogramm zu Bucket-Obergrenzen auf.
   3. Sortiere Eingabefeld in situ durch Ringtausch:
          Fuer jeden Bucket k = 1, 2,... :
              Solange noch Elemente > k im Bucket sind:
                  Fuehre Ringtausch durch bis Ausgangsposition wieder erreicht
      M.a.W.: Der Reihe nach wird jeder Bucket von fremden Elementen
      befreit.  Da Elemente < k nicht mehr vorhanden sein koennen,
      kann dies als Abbruchbedinung verwendet werden, so dass man
      keine Bucket- Untergrenzen speichern muss. *)


TYPE Key           = [1..5] ;  (* zulaessige Schluesselmenge *)
     KeyOrSentinel = [0..5] ;  (* Schluesselmenge mit Sentinelelement *)


PROCEDURE BucketSort(VAR a: ARRAY OF KeyOrSentinel) =
VAR count := ARRAY Key OF INTEGER {0,..} ;
    u, v : Key ;
BEGIN
    a[0] := FIRST(KeyOrSentinel) ; (* Sentinel *)

    (* Histogramm erzeugen *)
    FOR i := 1 TO LAST(a) DO
        INC(count[a[i]]) ;
    END ;

    (* Histogramm in Indizes umrechnen *)
    FOR k := FIRST(Key) + 1 TO LAST(Key) DO
        count[k] := count[k] + count[k-1] ;
    END ;

    (* Sortieren *)
    FOR k := FIRST(Key) TO LAST(Key) DO
        WHILE a[count[k]] >= k DO
            v := a[count[k]] ;
            REPEAT (* Ringtausch *)
                u := v ;
                v := a[count[u]] ;
                a[count[u]] := u ;
                DEC(count[u]) ;
            UNTIL u = k ;
        END ;
    END ;
END BucketSort ;


VAR data : ARRAY [0..50] OF KeyOrSentinel ;
BEGIN
    FOR i := 1 TO LAST(data) DO
        data[i] := VAL(Random(5), Key) ;
        IO.PutInt(data[i]) ; IO.Put("\n") ;
    END ;
    IO.Put("\n") ;

    BucketSort(data) ;

    FOR i := 1 TO LAST(data) DO
        IO.PutInt(data[i]) ; IO.Put("\n") ;
    END ;
END Main .
```

May 30, 00 11:10 **Main.m3** Page 1/1

```
MODULE Main ;

IMPORT IO ;
FROM Aufgabe35 IMPORT Random ;


TYPE Item = INTEGER ;


PROCEDURE Swap(VAR a, b : Item) =
VAR t : Item ;
BEGIN
    t := a ; a := b ; b := t ;
END Swap ;


(* Elemete von a zufaellig permutieren.
   Idee: "SelectionSort" mit zufaelligem Auswahlschritt *)

PROCEDURE Permute(VAR a : ARRAY OF Item) =
BEGIN
    FOR i := FIRST(a) TO LAST(a) - 1 DO
        Swap(a[i], a[i + Random(LAST(a) - i)]) ;
    END ;
END Permute ;


(* Hauptprogramm einige Permutationen der Zahlen von 1 bis 20 ausgeben *)

VAR a : ARRAY [1..20] OF Item ;

BEGIN
    FOR i := FIRST(a) TO LAST(a) DO
        a[i] := i ;
    END ;

    FOR j := 1 TO 10 DO
        Permute(a) ;

        FOR i := FIRST(a) TO LAST(a) DO
            IO.PutInt(a[i]) ;
            IO.Put(" ") ;
        END ;
        IO.Put("\n") ;
    END ;
END Main .
```

May 30, 00 11:10 **output.txt** Page

```
18 1 4 2 15 11 12 7 5 6 3 10 8 9 13 17 14 20 16 19
16 5 12 13 10 17 6 2 9 1 20 11 7 3 15 19 4 18 8 14
2 18 8 6 15 10 9 11 16 19 5 17 14 20 3 4 13 12 7 1
13 16 11 7 12 20 5 15 2 1 8 9 10 3 6 18 17 19 4 14
17 19 15 5 9 6 1 20 13 16 7 2 4 11 14 3 8 18 12 10
16 13 3 18 5 7 20 14 8 2 9 11 12 6 15 10 17 1 19 4
12 8 6 15 7 13 4 5 3 14 1 18 20 9 16 19 2 10 11 17
16 3 11 6 18 7 2 13 14 19 10 8 9 1 17 4 20 12 5 15
3 12 10 7 8 1 19 18 15 6 16 9 5 11 13 17 2 14 20 4
11 15 5 8 13 9 6 7 1 18 14 17 20 19 12 16 3 10 4 2
```

```
INTERFACE LinkedList ;

PROCEDURE Init();

PROCEDURE ToFirst() ;
PROCEDURE ToLast() ;
PROCEDURE Forth() ;
PROCEDURE Back() ;

PROCEDURE IsAfter() : BOOLEAN ;
PROCEDURE IsBefore() : BOOLEAN ;
PROCEDURE Get() : INTEGER ;
PROCEDURE Set(item: INTEGER) ;

PROCEDURE Append(item : INTEGER);
PROCEDURE InsertBefore(item : INTEGER);
PROCEDURE Delete();

END LinkedList.
```



```
MODULE LinkedList ;

TYPE Item = INTEGER ;
     Node = REF RECORD
         key        : Item ;
         prev, next : Node ;
     END ;

VAR  head, tail, current : Node ;


PROCEDURE Init() =
BEGIN
    head := NEW(Node) ;
    tail := NEW(Node) ;
    head^.prev := head ;
    head^.next := tail ;
    tail^.prev := head ;
    tail^.next := tail ;
    current := head ;
END Init ;


PROCEDURE ToFirst() =
BEGIN
    current := head^.next ;
END ToFirst ;


PROCEDURE ToLast() =
BEGIN
    current := tail^.prev ;
END ToLast ;


PROCEDURE IsAfter() : BOOLEAN =
BEGIN
    RETURN current = tail ;
END IsAfter ;

PROCEDURE IsBefore() : BOOLEAN =
BEGIN
    RETURN current = head ;
END IsBefore ;


PROCEDURE Forth() =
BEGIN
    current := current^.next ;
END Forth ;


PROCEDURE Back() =
BEGIN
    current := current^.prev ;
END Back ;


PROCEDURE Get() : INTEGER =
BEGIN
    <* ASSERT NOT (IsBefore() OR IsAfter()) *>

    RETURN current^.key ;
END Get ;
```

```
    PROCEDURE Set(item: INTEGER) =
    BEGIN
        <* ASSERT NOT (IsBefore() OR IsAfter()) *>

        current^.key := item ;
    END Set ;


    PROCEDURE Append(item : INTEGER) =
    VAR n : Node ;
    BEGIN
        n := NEW(Node) ;
        n^.prev := tail^.prev ;
        n^.next := tail ;
        n^.prev^.next := n ;
        n^.next^.prev := n ;
        n^.key := item ;
    END Append ;


    PROCEDURE InsertBefore(item : INTEGER) =
    VAR n : Node ;
    BEGIN
        <* ASSERT NOT IsBefore() *>

        n := NEW(Node) ;
        n^.prev := current^.prev ;
        n^.next := current ;
        n^.prev^.next := n ;
        n^.next^.prev := n ;
        n^.key := item ;
    END InsertBefore ;


    PROCEDURE Delete() =
    BEGIN
        <* ASSERT NOT (IsBefore() OR IsAfter()) *>

        current^.next^.prev := current^.prev ;
        current^.prev^.next := current^.next ;
        current := current^.next ;
    END Delete ;

    BEGIN
    END LinkedList .
```



```
    MODULE Main ;

    IMPORT IO ;
    IMPORT LinkedList AS List ;


    PROCEDURE WriteList() =
    BEGIN
        List.ToFirst() ;
        WHILE NOT List.IsAfter() DO
            IO.PutInt(List.Get()) ;
            IO.Put("\t") ;
            List.Forth() ;
        END ;
        IO.Put("\n") ;
    END WriteList ;


    BEGIN
        List.Init() ;
        List.Append(1) ;
        List.Append(2) ;
        List.Append(3) ;
        List.Append(4) ;
        List.Append(5) ;

        WriteList() ;

        List.ToFirst() ;
        List.Forth() ;
        List.InsertBefore(6) ;

        WriteList() ;

        List.ToLast() ;
        List.Back() ;
        List.Delete() ;

        WriteList() ;
    END Main .
```

```
INTERFACE LinkedList ;

PROCEDURE Init();

PROCEDURE ToFirst() ;
PROCEDURE ToLast() ;
PROCEDURE Forth() ;
PROCEDURE Back() ;

PROCEDURE IsAfter() : BOOLEAN ;
PROCEDURE IsBefore() : BOOLEAN ;
PROCEDURE Get() : INTEGER ;
PROCEDURE Set(item: INTEGER) ;

PROCEDURE Append(item : INTEGER);
PROCEDURE InsertBefore(item : INTEGER);
PROCEDURE Delete();

END LinkedList.
```



```
MODULE LinkedList ;

TYPE Item = INTEGER ;
     Node = REF RECORD
         key        : Item ;
         prev, next : Node ;
     END ;

VAR  head, tail, current : Node ;


PROCEDURE Init() =
BEGIN
    head := NEW(Node) ;
    tail := NEW(Node) ;
    head^.prev := head ;
    head^.next := tail ;
    tail^.prev := head ;
    tail^.next := tail ;
    current := head ;
END Init ;


PROCEDURE ToFirst() =
BEGIN
    current := head^.next ;
END ToFirst ;


PROCEDURE ToLast() =
BEGIN
    current := tail^.prev ;
END ToLast ;


PROCEDURE IsAfter() : BOOLEAN =
BEGIN
    RETURN current = tail ;
END IsAfter ;

PROCEDURE IsBefore() : BOOLEAN =
BEGIN
    RETURN current = head ;
END IsBefore ;


PROCEDURE Forth() =
BEGIN
    current := current^.next ;
END Forth ;


PROCEDURE Back() =
BEGIN
    current := current^.prev ;
END Back ;


PROCEDURE Get() : INTEGER =
BEGIN
    <* ASSERT NOT (IsBefore() OR IsAfter()) *>

    RETURN current^.key ;
END Get ;
```

```
    PROCEDURE Set(item: INTEGER) =
    BEGIN
        <* ASSERT NOT (IsBefore() OR IsAfter()) *>

        current^.key := item ;
    END Set ;


    PROCEDURE Append(item : INTEGER) =
    VAR n : Node ;
    BEGIN
        n := NEW(Node) ;
        n^.prev := tail^.prev ;
        n^.next := tail ;
        n^.prev^.next := n ;
        n^.next^.prev := n ;
        n^.key := item ;
    END Append ;


    PROCEDURE InsertBefore(item : INTEGER) =
    VAR n : Node ;
    BEGIN
        <* ASSERT NOT IsBefore() *>

        n := NEW(Node) ;
        n^.prev := current^.prev ;
        n^.next := current ;
        n^.prev^.next := n ;
        n^.next^.prev := n ;
        n^.key := item ;
    END InsertBefore ;


    PROCEDURE Delete() =
    BEGIN
        <* ASSERT NOT (IsBefore() OR IsAfter()) *>

        current^.next^.prev := current^.prev ;
        current^.prev^.next := current^.next ;
        current := current^.next ;
    END Delete ;

    BEGIN
    END LinkedList .
```



```
    MODULE Main ;

    IMPORT IO ;
    IMPORT LinkedList AS List ;


    PROCEDURE WriteList() =
    BEGIN
        List.ToFirst() ;
        WHILE NOT List.IsAfter() DO
            IO.PutInt(List.Get()) ;
            IO.Put("\t") ;
            List.Forth() ;
        END ;
        IO.Put("\n") ;
    END WriteList ;


    BEGIN
        List.Init() ;
        List.Append(1) ;
        List.Append(2) ;
        List.Append(3) ;
        List.Append(4) ;
        List.Append(5) ;

        WriteList() ;

        List.ToFirst() ;
        List.Forth() ;
        List.InsertBefore(6) ;

        WriteList() ;

        List.ToLast() ;
        List.Back() ;
        List.Delete() ;

        WriteList() ;
    END Main .
```

May 02, 00 16:13 **SearchTree.i3** Page 1/1

```
    INTERFACE SearchTree ;

    TYPE Item = INTEGER ;
         Node = REF RECORD
             key         : Item ;
             left, right : Node ;
         END ;

    VAR root : Node ;

    PROCEDURE Init() ;
    PROCEDURE Insert(key : Item) ;
    PROCEDURE Search(key : Item) : BOOLEAN ;
    PROCEDURE Write() ;

    END SearchTree .
```

May 02, 00 16:13 **SearchTree.m3** Page

```
    MODULE SearchTree ;

    IMPORT IO ;


    PROCEDURE Init() =
    BEGIN
        root := NIL ;
    END Init ;


    PROCEDURE Insert(key : Item) =
    VAR p, n : Node ;
    BEGIN
        n := NEW(Node) ;
        n^.key   := key ;
        n^.left  := NIL ;
        n^.right := NIL ;

        IF root = NIL THEN
            root := n ;
        ELSE
            p := root ;
            LOOP
                IF key <= p^.key THEN
                    IF p^.left = NIL THEN
                        p^.left := n ;
                        RETURN ;
                    ELSE
                        p := p^.left ;
                    END ;
                ELSE (* key > p^.key *)
                    IF p^.right = NIL THEN
                        p^.right := n ;
                        RETURN ;
                    ELSE
                        p := p^.right ;
                    END ;
                END ;
            END ;
        END ;
    END Insert ;


    PROCEDURE Search(key : Item) : BOOLEAN =
    VAR n : Node ;
    BEGIN
        n := root ;
        WHILE n # NIL DO
            IF key = n^.key THEN
                RETURN TRUE ;
            ELSIF key <= n^.key THEN
                n := n^.left ;
            ELSE (* key > n^.key *)
                n := n^.right ;
            END ;
        END ;
        RETURN FALSE ;
    END Search ;


    PROCEDURE WriteSubTree(n : Node) =
    BEGIN
        IF n # NIL THEN
            WriteSubTree(n^.left) ;
```

```
            IO.PutInt(n^.key) ;
            IO.Put("\n") ;

            WriteSubTree(n^.right) ;
        END;
    END WriteSubTree ;


    PROCEDURE Write() =
    BEGIN
        WriteSubTree(root) ;
    END Write ;



    BEGIN
    END SearchTree .
```



```
    MODULE Main ;

    IMPORT IO, SearchTree, Traversal ;


    VAR i : INTEGER ;
    BEGIN
        SearchTree.Init() ;

        WHILE NOT IO.EOF() DO
            TRY
                i := IO.GetInt() ;
            EXCEPT
              IO.Error => EXIT ;
            END ;
            SearchTree.Insert(i) ;
        END ;

        IO.Put("Rekursive InOrder-Traversierung:\n") ;
        SearchTree.Write() ;

        IO.Put("\n\nlineares Durchsuchen:\n") ;
        FOR i := -1000 TO 1000 DO
            IF SearchTree.Search(i) THEN
                IO.PutInt(i) ;
                IO.Put("\n") ;
            END ;
        END ;

        IO.Put("\n\nPreOrder-Traversierung:\n") ;
        Traversal.TraversePreOrder() ;
        IO.Put("\n\nInOrder-Traversierung:\n") ;
        Traversal.TraverseInOrder() ;
        IO.Put("\n\nPostOrder-Traversierung:\n") ;
        Traversal.TraversePostOrder() ;
        IO.Put("\n\nLevelOrder-Traversierung:\n") ;
        Traversal.TraverseLevelOrder() ;
    END Main .
```

May 02, 00 17:22 **Stack.i3** Page 1/1

```
INTERFACE Stack ;

IMPORT SearchTree ;


CONST Capacity = 100 ;

TYPE Item = RECORD
    node : SearchTree.Node ;
    flag : BOOLEAN ;
END ;

PROCEDURE Push(item : Item) ;
PROCEDURE Pop() : Item ;
PROCEDURE IsFull() : BOOLEAN ;
PROCEDURE IsEmpty() : BOOLEAN ;

END Stack .
```

May 02, 00 16:20 **Stack.m3** Page

```
MODULE Stack ;

TYPE Index = [0..Capacity] ;

VAR store : ARRAY Index OF Item ;
    top   : Index ;


PROCEDURE Push(item : Item) =
BEGIN
    <* ASSERT NOT IsFull() *>
    store[top] := item ;
    INC(top) ;
END Push ;


PROCEDURE Pop() : Item =
BEGIN
    <* ASSERT NOT IsEmpty() *>
    DEC(top) ;
    RETURN store[top] ;
END Pop ;


PROCEDURE IsFull() : BOOLEAN =
BEGIN
    RETURN top > Capacity ;
END IsFull ;


PROCEDURE IsEmpty() : BOOLEAN =
BEGIN
    RETURN top <= 0 ;
END IsEmpty ;


BEGIN
    top := 0 ;
END Stack .
```

May 02, 00 15:47 **Queue.i3** Page 1/1

```
INTERFACE Queue ;

IMPORT SearchTree ;


CONST Capacity = 100 ;

TYPE Item = SearchTree.Node ;

PROCEDURE Put(item : Item) ;
PROCEDURE Get() : Item ;
PROCEDURE IsFull() : BOOLEAN ;
PROCEDURE IsEmpty() : BOOLEAN ;

END Queue .
```

Apr 28, 00 17:03 **Queue.m3** Page

```
MODULE Queue ;

TYPE Index = [0..Capacity] ;

VAR store      : ARRAY Index OF Item ;
    head, tail : Index ;


PROCEDURE Put(item : Item) =
BEGIN
    <* ASSERT NOT IsFull() *>

    store[tail] := item ;

    IF (tail < Capacity) THEN
        INC(tail) ;
    ELSE
        tail := 0 ;
    END ;
END Put ;


PROCEDURE Get() : Item =
VAR item : Item ;
BEGIN
    <* ASSERT NOT IsEmpty() *>

    item := store[head] ;

    IF (head < Capacity) THEN
        INC(head) ;
    ELSE
        head := 0 ;
    END ;

    RETURN item ;
END Get ;


PROCEDURE IsFull() : BOOLEAN =
BEGIN
    IF head = 0 THEN
        RETURN tail = Capacity ;
    ELSE
        RETURN tail = head - 1 ;
    END ;
END IsFull ;


PROCEDURE IsEmpty() : BOOLEAN =
BEGIN
    RETURN head = tail ;
END IsEmpty ;


BEGIN
    head := 0 ;
    tail := 0 ;
END Queue .
```

Apr 28, 00 15:27 **Traversal.i3** Page 1/1

```
    INTERFACE Traversal ;

    PROCEDURE TraversePreOrder() ;
    PROCEDURE TraversePostOrder() ;
    PROCEDURE TraverseInOrder() ;
    PROCEDURE TraverseLevelOrder() ;

    END Traversal .
```

May 02, 00 17:22 **Traversal.m3** Page

```
    MODULE Traversal ;

    IMPORT IO, Stack, Queue ;
    FROM SearchTree IMPORT Node, root ;


    PROCEDURE Visit(n : Node) =
    BEGIN
        IO.PutInt(n^.key) ;
        IO.Put("\n") ;
    END Visit ;


    PROCEDURE TraversePreOrder() =
    VAR n : Node ;
    BEGIN
        Stack.Push(Stack.Item{root, TRUE}) ;
        WHILE NOT Stack.IsEmpty() DO
            n := Stack.Pop().node ;
            IF n # NIL THEN
                Visit(n) ;
                Stack.Push(Stack.Item{n^.right, TRUE}) ;
                Stack.Push(Stack.Item{n^.left,  TRUE}) ;
            END ;
        END ;
    END TraversePreOrder ;


    PROCEDURE TraverseInOrder() =
    VAR s : Stack.Item ;
    BEGIN
        Stack.Push(Stack.Item{root, TRUE}) ;
        WHILE NOT Stack.IsEmpty() DO
            s := Stack.Pop() ;
            IF s.node # NIL THEN
                IF s.flag THEN
                    Stack.Push(Stack.Item{s.node, FALSE}) ;
                    Stack.Push(Stack.Item{s.node^.left, TRUE}) ;
                ELSE
                    Visit(s.node) ;
                    Stack.Push(Stack.Item{s.node^.right, TRUE}) ;
                END ;
            END ;
        END ;
    END TraverseInOrder ;


    PROCEDURE TraversePostOrder() =
    VAR s : Stack.Item ;
    BEGIN
        Stack.Push(Stack.Item{root, TRUE}) ;
        WHILE NOT Stack.IsEmpty() DO
            s := Stack.Pop() ;
            IF s.node # NIL THEN
                IF s.flag THEN
                    Stack.Push(Stack.Item{s.node, FALSE}) ;
                    Stack.Push(Stack.Item{s.node^.right, TRUE}) ;
                    Stack.Push(Stack.Item{s.node^.left, TRUE}) ;
                ELSE (* s.flag = 1 *)
                    Visit(s.node) ;
                END ;
            END ;
        END ;
    END TraversePostOrder ;

```

```
    PROCEDURE TraverseLevelOrder() =
    VAR n : Node ;
    BEGIN
70      Queue.Put(root) ;
        WHILE NOT Queue.IsEmpty() DO
            n := Queue.Get() ;
            IF n # NIL THEN
                Visit(n) ;
75              Queue.Put(n^.left) ;
                Queue.Put(n^.right) ;
            END ;
        END ;
    END TraverseLevelOrder ;
80

    BEGIN
    END Traversal .
```



```
    Rekursive InOrder-Traversierung:
    -798
    -611
    -495
5   -299
    -222
    -125
    -105
    -100
10  -89
    -34
    -30
    -30
    -8
15  -5
    -3
    2
    10
    13
20  18
    20
    100
    105
    121
25  200
    300
    450
    500
    501
30

    lineares Durchsuchen:
    -798
    -611
35  -495
    -299
    -222
    -125
    -105
40  -100
    -89
    -34
    -30
    -8
45  -5
    -3
    2
    10
    13
50  18
    20
    100
    105
    121
55  200
    300
    450
    500
    501
60

    PreOrder-Traversierung:
    18
    -5
65  -8
    -30
```

```
-89
-100
-105
-611
-798
-222
-299
-495
-125
-34
-30
-3
10
2
13
121
100
20
105
200
300
500
450
501


InOrder-Traversierung:
-798
-611
-495
-299
-222
-125
-105
-100
-89
-34
-30
-30
-8
-5
-3
2
10
13
18
20
100
105
121
200
300
450
500
501


PostOrder-Traversierung:
-798
-495
-299
-125
-222
-611
-105
-100
```



```
-30
-34
-89
-30
-8
2
13
10
-3
-5
20
105
100
450
501
500
300
200
121
18


LevelOrder-Traversierung:
18
-5
121
-8
-3
100
200
-30
10
20
105
300
-89
2
13
500
-100
-34
450
501
-105
-30
-611
-798
-222
-299
-125
-495
```

May 02, 00 16:13 **SearchTree.i3** Page 1/1

```
    INTERFACE SearchTree ;

    TYPE Item = INTEGER ;
         Node = REF RECORD
             key         : Item ;
             left, right : Node ;
         END ;

    VAR root : Node ;

    PROCEDURE Init() ;
    PROCEDURE Insert(key : Item) ;
    PROCEDURE Search(key : Item) : BOOLEAN ;
    PROCEDURE Write() ;

    END SearchTree .
```

May 02, 00 16:13 **SearchTree.m3** Page

```
    MODULE SearchTree ;

    IMPORT IO ;


    PROCEDURE Init() =
    BEGIN
        root := NIL ;
    END Init ;


    PROCEDURE Insert(key : Item) =
    VAR p, n : Node ;
    BEGIN
        n := NEW(Node) ;
        n^.key   := key ;
        n^.left  := NIL ;
        n^.right := NIL ;

        IF root = NIL THEN
            root := n ;
        ELSE
            p := root ;
            LOOP
                IF key <= p^.key THEN
                    IF p^.left = NIL THEN
                        p^.left := n ;
                        RETURN ;
                    ELSE
                        p := p^.left ;
                    END ;
                ELSE (* key > p^.key *)
                    IF p^.right = NIL THEN
                        p^.right := n ;
                        RETURN ;
                    ELSE
                        p := p^.right ;
                    END ;
                END ;
            END ;
        END ;
    END Insert ;


    PROCEDURE Search(key : Item) : BOOLEAN =
    VAR n : Node ;
    BEGIN
        n := root ;
        WHILE n # NIL DO
            IF key = n^.key THEN
                RETURN TRUE ;
            ELSIF key <= n^.key THEN
                n := n^.left ;
            ELSE (* key > n^.key *)
                n := n^.right ;
            END ;
        END ;
        RETURN FALSE ;
    END Search ;


    PROCEDURE WriteSubTree(n : Node) =
    BEGIN
        IF n # NIL THEN
            WriteSubTree(n^.left) ;
```

```
            IO.PutInt(n^.key) ;
            IO.Put("\n") ;

            WriteSubTree(n^.right) ;
        END;
    END WriteSubTree ;


    PROCEDURE Write() =
    BEGIN
        WriteSubTree(root) ;
    END Write ;



    BEGIN
    END SearchTree .
```



```
    MODULE Main ;

    IMPORT IO, SearchTree, Traversal ;


    VAR i : INTEGER ;
    BEGIN
        SearchTree.Init() ;

        WHILE NOT IO.EOF() DO
            TRY
                i := IO.GetInt() ;
            EXCEPT
              IO.Error => EXIT ;
            END ;
            SearchTree.Insert(i) ;
        END ;

        IO.Put("Rekursive InOrder-Traversierung:\n") ;
        SearchTree.Write() ;

        IO.Put("\n\nlineares Durchsuchen:\n") ;
        FOR i := -1000 TO 1000 DO
            IF SearchTree.Search(i) THEN
                IO.PutInt(i) ;
                IO.Put("\n") ;
            END ;
        END ;

        IO.Put("\n\nPreOrder-Traversierung:\n") ;
        Traversal.TraversePreOrder() ;
        IO.Put("\n\nInOrder-Traversierung:\n") ;
        Traversal.TraverseInOrder() ;
        IO.Put("\n\nPostOrder-Traversierung:\n") ;
        Traversal.TraversePostOrder() ;
        IO.Put("\n\nLevelOrder-Traversierung:\n") ;
        Traversal.TraverseLevelOrder() ;
    END Main .
```

May 02, 00 17:22 **Stack.i3** Page 1/1

```
INTERFACE Stack ;

IMPORT SearchTree ;


CONST Capacity = 100 ;

TYPE Item = RECORD
    node : SearchTree.Node ;
    flag : BOOLEAN ;
END ;

PROCEDURE Push(item : Item) ;
PROCEDURE Pop() : Item ;
PROCEDURE IsFull() : BOOLEAN ;
PROCEDURE IsEmpty() : BOOLEAN ;

END Stack .
```

May 02, 00 16:20 **Stack.m3** Page

```
MODULE Stack ;

TYPE Index = [0..Capacity] ;

VAR store : ARRAY Index OF Item ;
    top   : Index ;


PROCEDURE Push(item : Item) =
BEGIN
    <* ASSERT NOT IsFull() *>
    store[top] := item ;
    INC(top) ;
END Push ;


PROCEDURE Pop() : Item =
BEGIN
    <* ASSERT NOT IsEmpty() *>
    DEC(top) ;
    RETURN store[top] ;
END Pop ;


PROCEDURE IsFull() : BOOLEAN =
BEGIN
    RETURN top > Capacity ;
END IsFull ;


PROCEDURE IsEmpty() : BOOLEAN =
BEGIN
    RETURN top <= 0 ;
END IsEmpty ;


BEGIN
    top := 0 ;
END Stack .
```

May 02, 00 15:47 **Queue.i3** Page 1/1

```
INTERFACE Queue ;

IMPORT SearchTree ;


CONST Capacity = 100 ;

TYPE Item = SearchTree.Node ;

PROCEDURE Put(item : Item) ;
PROCEDURE Get() : Item ;
PROCEDURE IsFull() : BOOLEAN ;
PROCEDURE IsEmpty() : BOOLEAN ;

END Queue .
```

Apr 28, 00 17:03 **Queue.m3** Page

```
MODULE Queue ;

TYPE Index = [0..Capacity] ;

VAR store      : ARRAY Index OF Item ;
    head, tail : Index ;


PROCEDURE Put(item : Item) =
BEGIN
    <* ASSERT NOT IsFull() *>

    store[tail] := item ;

    IF (tail < Capacity) THEN
        INC(tail) ;
    ELSE
        tail := 0 ;
    END ;
END Put ;


PROCEDURE Get() : Item =
VAR item : Item ;
BEGIN
    <* ASSERT NOT IsEmpty() *>

    item := store[head] ;

    IF (head < Capacity) THEN
        INC(head) ;
    ELSE
        head := 0 ;
    END ;

    RETURN item ;
END Get ;


PROCEDURE IsFull() : BOOLEAN =
BEGIN
    IF head = 0 THEN
        RETURN tail = Capacity ;
    ELSE
        RETURN tail = head - 1 ;
    END ;
END IsFull ;


PROCEDURE IsEmpty() : BOOLEAN =
BEGIN
    RETURN head = tail ;
END IsEmpty ;


BEGIN
    head := 0 ;
    tail := 0 ;
END Queue .
```

Apr 28, 00 15:27 **Traversal.i3** Page 1/1

```
    INTERFACE Traversal ;

    PROCEDURE TraversePreOrder() ;
    PROCEDURE TraversePostOrder() ;
    PROCEDURE TraverseInOrder() ;
    PROCEDURE TraverseLevelOrder() ;

    END Traversal .
```

May 02, 00 17:22 **Traversal.m3** Page

```
    MODULE Traversal ;

    IMPORT IO, Stack, Queue ;
    FROM SearchTree IMPORT Node, root ;


    PROCEDURE Visit(n : Node) =
    BEGIN
        IO.PutInt(n^.key) ;
        IO.Put("\n") ;
    END Visit ;


    PROCEDURE TraversePreOrder() =
    VAR n : Node ;
    BEGIN
        Stack.Push(Stack.Item{root, TRUE}) ;
        WHILE NOT Stack.IsEmpty() DO
            n := Stack.Pop().node ;
            IF n # NIL THEN
                Visit(n) ;
                Stack.Push(Stack.Item{n^.right, TRUE}) ;
                Stack.Push(Stack.Item{n^.left,  TRUE}) ;
            END ;
        END ;
    END TraversePreOrder ;


    PROCEDURE TraverseInOrder() =
    VAR s : Stack.Item ;
    BEGIN
        Stack.Push(Stack.Item{root, TRUE}) ;
        WHILE NOT Stack.IsEmpty() DO
            s := Stack.Pop() ;
            IF s.node # NIL THEN
                IF s.flag THEN
                    Stack.Push(Stack.Item{s.node, FALSE}) ;
                    Stack.Push(Stack.Item{s.node^.left, TRUE}) ;
                ELSE
                    Visit(s.node) ;
                    Stack.Push(Stack.Item{s.node^.right, TRUE}) ;
                END ;
            END ;
        END ;
    END TraverseInOrder ;


    PROCEDURE TraversePostOrder() =
    VAR s : Stack.Item ;
    BEGIN
        Stack.Push(Stack.Item{root, TRUE}) ;
        WHILE NOT Stack.IsEmpty() DO
            s := Stack.Pop() ;
            IF s.node # NIL THEN
                IF s.flag THEN
                    Stack.Push(Stack.Item{s.node, FALSE}) ;
                    Stack.Push(Stack.Item{s.node^.right, TRUE}) ;
                    Stack.Push(Stack.Item{s.node^.left, TRUE}) ;
                ELSE (* s.flag = 1 *)
                    Visit(s.node) ;
                END ;
            END ;
        END ;
    END TraversePostOrder ;

```

```
    PROCEDURE TraverseLevelOrder() =
    VAR n : Node ;
    BEGIN
70      Queue.Put(root) ;
        WHILE NOT Queue.IsEmpty() DO
            n := Queue.Get() ;
            IF n # NIL THEN
                Visit(n) ;
75              Queue.Put(n^.left) ;
                Queue.Put(n^.right) ;
            END ;
        END ;
    END TraverseLevelOrder ;
80

    BEGIN
    END Traversal .
```



```
    Rekursive InOrder-Traversierung:
    -798
    -611
    -495
5   -299
    -222
    -125
    -105
    -100
10  -89
    -34
    -30
    -30
    -8
15  -5
    -3
    2
    10
    13
20  18
    20
    100
    105
    121
25  200
    300
    450
    500
    501
30

    lineares Durchsuchen:
    -798
    -611
35  -495
    -299
    -222
    -125
    -105
40  -100
    -89
    -34
    -30
    -8
45  -5
    -3
    2
    10
    13
50  18
    20
    100
    105
    121
55  200
    300
    450
    500
    501
60

    PreOrder-Traversierung:
    18
    -5
65  -8
    -30
```

```
-89
-100
-105
-611
-798
-222
-299
-495
-125
-34
-30
-3
10
2
13
121
100
20
105
200
300
500
450
501


InOrder-Traversierung:
-798
-611
-495
-299
-222
-125
-105
-100
-89
-34
-30
-30
-8
-5
-3
2
10
13
18
20
100
105
121
200
300
450
500
501


PostOrder-Traversierung:
-798
-495
-299
-125
-222
-611
-105
-100
```



```
-30
-34
-89
-30
-8
2
13
10
-3
-5
20
105
100
450
501
500
300
200
121
18


LevelOrder-Traversierung:
18
-5
121
-8
-3
100
200
-30
10
20
105
300
-89
2
13
500
-100
-34
450
501
-105
-30
-611
-798
-222
-299
-125
-495
```